Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Stock Market Prediction

M11 Seminar: Deep Learning

February 20, 2024

Lecturer:
Dr. Alla Petukhina

Authors:
Karla Valeria Mondragón Rosas — 591102
Ruy Guzman Camacho — 591101
Fernanda de León — 591100

Index

1. Individual Stock Prediction

1.1. Introduction

The financial markets are famously complex and dynamic in nature, predicting stock behavior remains a challenging yet crucial task for investors and traders seeking to optimize their portfolios and make informed decisions.

In recent years, advancements in machine learning algorithms have offered promising methodologies for accurate stock price forecasting. This paper explores the application of three predictive models: Random Forest, XGBoost, and Long Short-Term Memory (LSTM) neural networks, to predict the behavior of stocks from three notable companies: Google, Microsoft, and Visa.

The research aims to assess and compare the performance and accuracy of these models in predicting stock behavior over time in a trading environment. The selected companies represent sectors characterized by distinct market dynamics, with limited volatility.

Through this research, we seek to identify the strengths and constraints of each model and we aim to contribute valuable insights to the field of financial forecasting with machine learning algorithms. This study holds the potential to inform investment strategies, improve risk management practices, and contribute to future development in the context of machine learning for financial markets.

1.2. Motivation

This study is incentivized by the methodologies outlined in the literature discussed in section **1.3. 'Related Work'**, with the main objective of replicating and expanding these approaches. Our research is particularly motivated by the ambition to compare the performance and accuracy of a non-tree model and a tree model, assessing their capability of capturing patterns within financial time-series data.

Additionally, another key objective is determining optimal buy or sell points to enable profitable and informed decision-making within the dynamic environment of the stock market.

1.3. Related Work

The content of this paper is based on the work realized on the following papers, which include examples of stock prediction as a classification problem and comparing Random Forest and XGBoost, and comparing the performance of LSTM vs ARIMA in economics environments for data in time series.

- Predicting the direction of stock market prices using tree-based classifiers. (Basak, Kar, Saha, Khaidem, & Dey)
- Forecasting Economics and Financial Time Series: ARIMA vs. LSTM. (Siami-Namini & Namin)

1.4. Challenge

The challenge within this study encompasses the implementation and comparison of three different predictive algorithms —Random Forest, XGBoost, and Long Short-Term Memory (LSTM) neural networks— across three notable companies: Google, Microsoft, and VISA. The objective is to assess the profitability of each model in a trading environment, given an initial investment of $10,000 USD for each stock, meaning $30,000 USD in total.

Each algorithm will be applied individually to the stock prices of Google, Microsoft, and VISA. The models will be implemented to the data to make predictions and inform investment decisions. The challenge lies in evaluating and comparing the performance and accuracy of these algorithms. The ultimate goal is to determine which model yields the most profitable outcomes based on the initial investment.

1.5. Dataset and Methodologies

The datasets used in the three models is a sample from **S&P 500** obtained via **Yahoo Finance** for three technology companies: **Microsoft, Google and Visa**. The data gathered represents data between the dates **01.01.2021** and **02.12.2023**. The

dataset contains the following columns: **Date, Open, High, Low, Close, Adj Close, Volume**

1.6. Feature Extraction

The LSTM implementation works using only the closing price of each individual stock. For both Random Forest and XGBoost we make the calculation of multiple momentum indicators that will describe output signals to buy or sell stock, therefore indicating tendencies.

Relative Strength Index

Momentum oscillator that measures the speed and change of price movements. The range of the RSI oscillates from zero to 100. The RSI is classified as oversold when it falls below 30 and overbought when it rises beyond 70.

$$RSI = 100 - \frac{100}{1+RS}$$

$$RS = \frac{AverageGainOverPast14Days}{AverageLossOverPast14Days}$$

- RS: Relative strength

Stochastic Oscillator

Momentum indicator that, over a predetermined number of periods, displays the closing price's location in relation to the high-low range. The indicator's possible values are 0 to 100. In a downtrend, the closing price usually closes to the low, while in an uptrend, it usually closes to the high.

$$\%k = 100 \times \frac{(C-L_{14})}{(H_{14}-L_{14})}$$

- C: Current Closing price
- L_{14}: Lowest price in the last 14 days
- H_{14}: Highest price in the last 14 days

Williams percentage rate

Momentum indicator to determine when to enter and leave a position. The values it

employs range from 0 to -100, where 0 denotes an overbought market and -100 denotes an oversold market.

$$\%R = -100 \times \frac{(H_{14} - C)}{(H_{14} - L_{14})}$$

- C: Current Closing price
- L_{14}: Lowest price in the last 14 days
- H_{14}: Highest price in the last 14 days

Moving Average Convergence Divergence

Momentum indicator that follows trends and displays the correlation between two exponential moving averages (EMAs) of the price of an asset. The 26-period EMA is subtracted from the 12-period EMA to get the MACD line. If the result is below the signal indicates a sell signal, otherwise it's a buy signal.

$$MACD = EMA_{12}(C) - EMA_{26}(C)$$

$$SignalLine = EMA_{9}(MACD)$$

- C: Current Closing price
- EMA_n: n-day exponential moving average

Price rate of change

Momentum indicator that measures the rate of change between the ongoing cost and the value of a certain number of periods ago. Represents how fast prices move.

$$PROC = \frac{C_t - C_{t-n}}{C_{t-n}}$$

- C_t: Current Closing price at time t
- C_{t-n}: Current Closing price at time t-n

On Balance Volume

Indicator, that estimates changes in price based on changes in the trading volume, determines buying and selling trends using the cumulative volume. The cumulative volume adds if the prices go up, and it subtracts the volume when prices go down.

$$OBV(t) = \begin{cases} OBV(t-1) + Vol(t) & \text{if } C(t) > C(t-1) \\ OBV(t-1) - Vol(t) & \text{if } C(t) < C(t-1) \\ OBV(t-1) & \text{if } C(t) = C(t-1) \end{cases}$$

- $OBV(t)$: On Balance Volume at time t
- $Vol(t)$: Trading Volume at time t
- $C(t)$: Closing price at time t

1.7 Target

The target for the model is “Direction Prediction” which represents the tendency for the stock price to increase, decrease or stay still in a period of time. Three values represent the direction.

Random Forest classes	XGBoost Classes
• 1 if positive • -1 if negative • 0 if no change	• 2 if positive, • 0 if negative • 1 if no change

Table 1.1 Target definition

2. Long Short-Term Memory (LSTM)

2.1. What is a neural network?

A neural network can be defined as a subset of machine learning, their name and structure inspired by the human brain since they intend to mimic how biological neurons signal each other. Neural networks consist of at least 3 node layers: an input layer, one or more hidden layers and an output layer. Each node or neuron is connected to another through links called “synapses” and has a weight and threshold associated (Siami‑Namini., Namin., 2018). The number of nodes in the input layers depends on the dataset and the number of features.

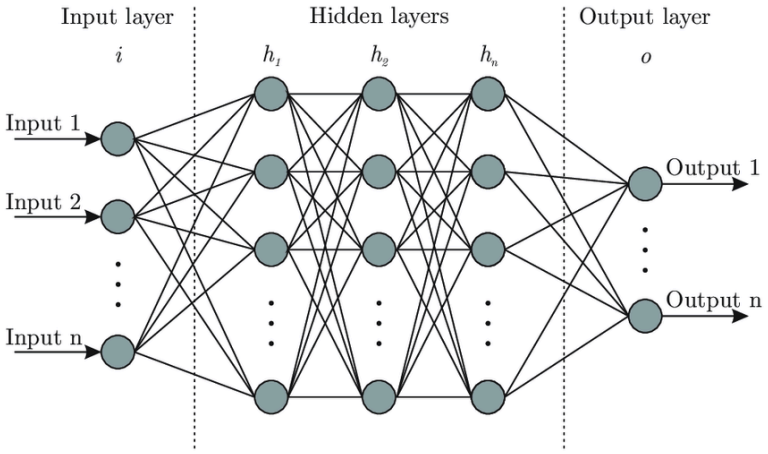


Figure 2.1. Neural Network[1]

Once an input layer is determined, the weights are assigned –larger ones contributing more significantly to the output compared to other inputs–. All inputs are then multiplied by their respective weights and then summed. The role of the weights can be described as the ‘decision maker’ to decide which input gets through and which doesn’t, is through the weight that a neural network basically learns by adjusting the weight for each synopsis (*What Are Neural Networks?, IBM*, n.d.).

Within the hidden layers an activation function is applied –either the sigmoid or the tangent hyperbolic, in most cases– on the weighted sum of inputs to transform them

[1] Shukla, L. (2023, July 18). Designing your neural networks - towards data science. *Medium*. https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed

into outputs, or predicted values. If this output exceeds the threshold, it activates the node, passing data to the next layer in the network (*What Are Neural Networks?, IBM*, n.d.). This procedure is repeated a number of times –epochs– with the same features and the weights are re-adjusted until there's an improvement in the predicted values and until the minimum error rate or cost is minimized –the difference between expected and predicted values–, using the *SoftMax* function. When the cost is minimized it's considered that the model has been trained (Siami-Namini., Namin., 2018).

2.1.1. What is a Recurrent Neural Network (RNN)?

A Recurrent Neural Network (RNN) is a type of neural network which uses sequential data or time-series data, where the objective is to predict the next step on a sequence, considering the previous steps observed in said sequence (*What Are Recurrent Neural Networks?* | *IBM*, n.d.).

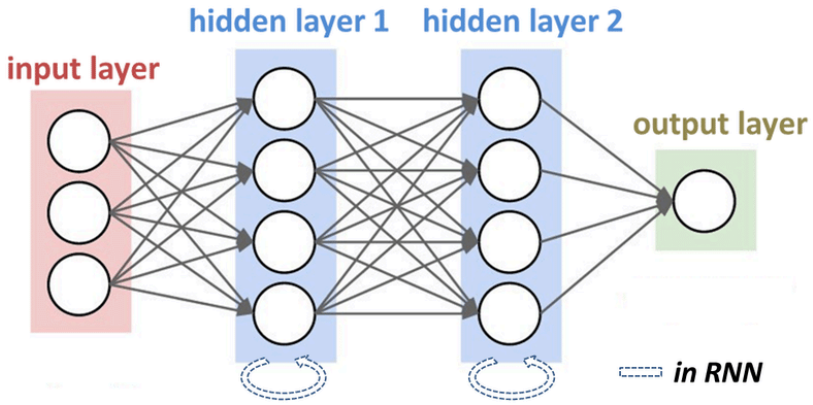


Figure 2.2. Recurrent Neural Network[2]

Recurrent neural networks are distinguished by their 'memory', since the sequential observations and earlier stages need to be remembered to forecast future trends. In contrast with traditional neural networks, RNNs don't assume that inputs and outputs are independent from each other, the output of a RNN depends on the prior elements of the sequence.

[2] Ma, S., Xiao, B., Hong, R., Addissie, B. D., Drikas, Z. B., Antonsen, T. M., Ott, E., & Anlage, S. M. (2019). Classification and Prediction of Wave Chaotic Systems with Machine Learning Techniques. Acta Physica Polonica A, 136(5), 757–764. https://doi.org/10.12693/aphyspola.136.757

In recurrent neural networks, the hidden layers act as internal storage for storing the information acquired in earlier stages of reading sequential data (Siami-Namini., Namin., 2018). RNNs are recurrent because they perform the same task on every element of the sequence and because these networks remember only a few earlier steps in the sequence – thus are not suitable to remember longer sequences of data – this derives in two challenges “exploding gradients” and “vanishing gradients”. These both issues are defined by the size of the gradient or slope of the loss function along the error curve. Vanishing gradients occur, when the slope is too small, it keeps getting smaller throughout the algorithm, updating the weight parameter until it becomes insignificant causing the algorithm to no longer learn. On the other hand, exploding gradients manifest when the slope is too large creating an unstable model, when this happens the model weights grow too large, and will eventually be represented as NaN (*What Are Recurrent Neural Networks?* | *IBM*, n.d.).

One solution to the problems mentioned above is to reduce the number of hidden layers, eliminating some of the complexity of the model. Another solution is to use a ‘memory line’, which is introduced in the Long Short-Term Memory (LSTM) recurrent network.

2.2. LSTM: How does it work?

LSTM is a kind of Recurrent Neural Network with additional features that allow the network to memorize the sequence of data. LSTM introduces a memory cell which works as a container that can hold information for an extended period and is controlled by three gates: forget gate, input gate and output gate.

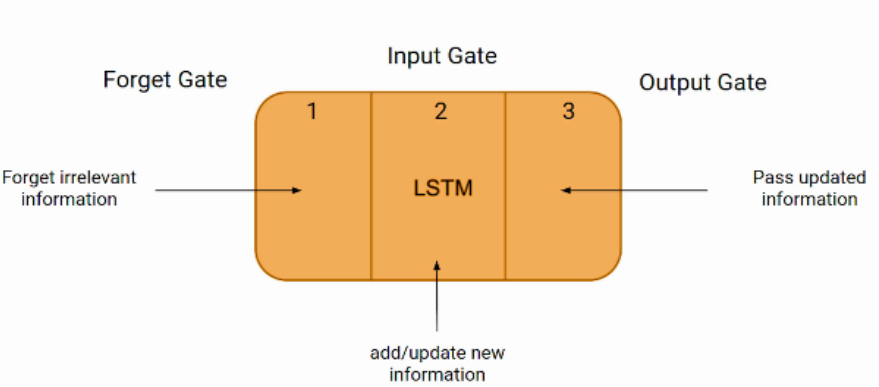


Figure 2.3. Long Short-Term Memory Neural Network's Gates[3]

- **Forget Gate:** Controls what information is removed from the memory cell. Outputs a number between 0 and 1, where 1 means 'keep' and 0 implies 'ignore' (GfG, 2023).
- **Input Gate:** Controls what information is stored in the memory cell.
    - First, the information is regulated using the sigmoid function and filters the values to be remembered returning a value between 0 and 1 (GfG, 2023).

$$f(x) = \frac{1}{e^{x}+1}$$

    - Next, a vector is created using the TanH function that gives an output from -1 to +1.

$$TanH(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{x}}$$

- **Output Gate:** Controls what information is output from the memory cell. The yielded value will be based on the cell state along with the filtered and newly added data (GfG, 2023).

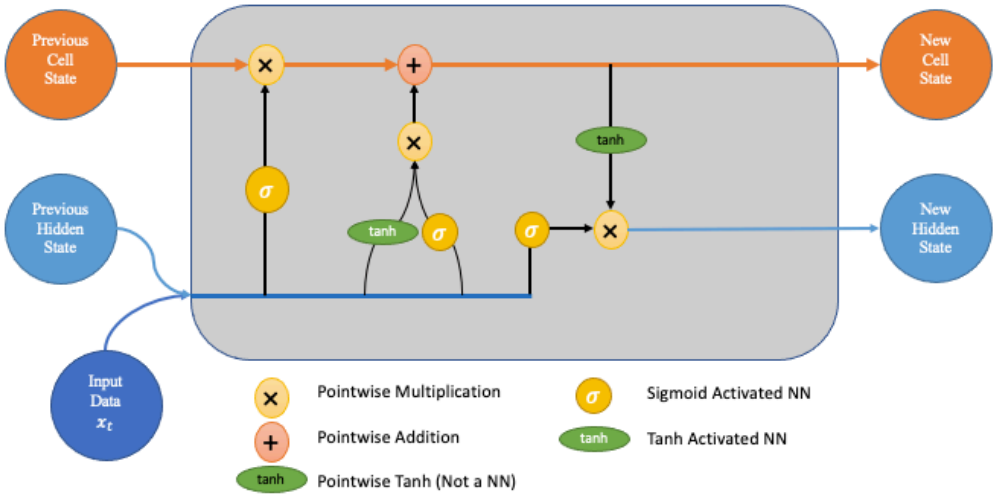


Figure 2.4. Long Short-Term Memory Neural Network Memory Cell[4]

[3] Saxena, S. (2024, January 4). What is LSTM? Introduction to Long Short-Term Memory. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

[4] Dolphin, R. (2022, February 28). LSTM Networks | A detailed explanation | towards Data science. Medium. https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9

Each LSTM is a set of cells where the data streams are captured and stored. The cells resemble a transport line (the upper line in each cell) that connects out of one module to another one conveying data from the past and gathering them for the present one. Due to the use of some gates in each cell, data in each cell can be disposed of, filtered, or added for the next cells (Siami-Namini., Namin., 2018).

2.2.1. Why use LSTM?

LSTM are considered suitable for stock market prediction, primarily due to their ability to capture and model complex sequential dependencies in time-series data. Since the stock market presents some temporal dependencies i.e. the current price of a stock often depends on past prices, the memory line of LSTM networks are well-suited to capture and leverage the temporal patterns present in stock price sequences. Moreover, the financial market can be unpredictable and LSTMs are known for their robustness to tangenital data, and their ability to filter out irrelevant information helps in focusing on the most influential patterns.

2.3. LSTM in Action

2.3.1. Data Visualization

The implementation starts with visualizing the dataframes, one for each company. After dropping unnecessary variables, what is left is the ‘Date’ variable and with the feature ‘Close’ which refers to the closing price of the stock on a specific day.
The following figures demonstrate the behavior of the stock value of each company across time, starting from January 2022 until January 2024.

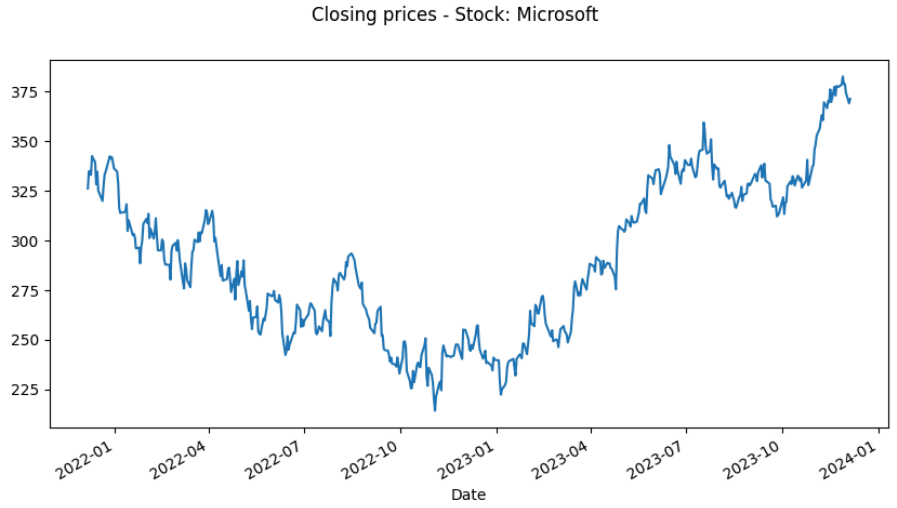


Figure 2.5. Closing Prices throughout time for Microsoft

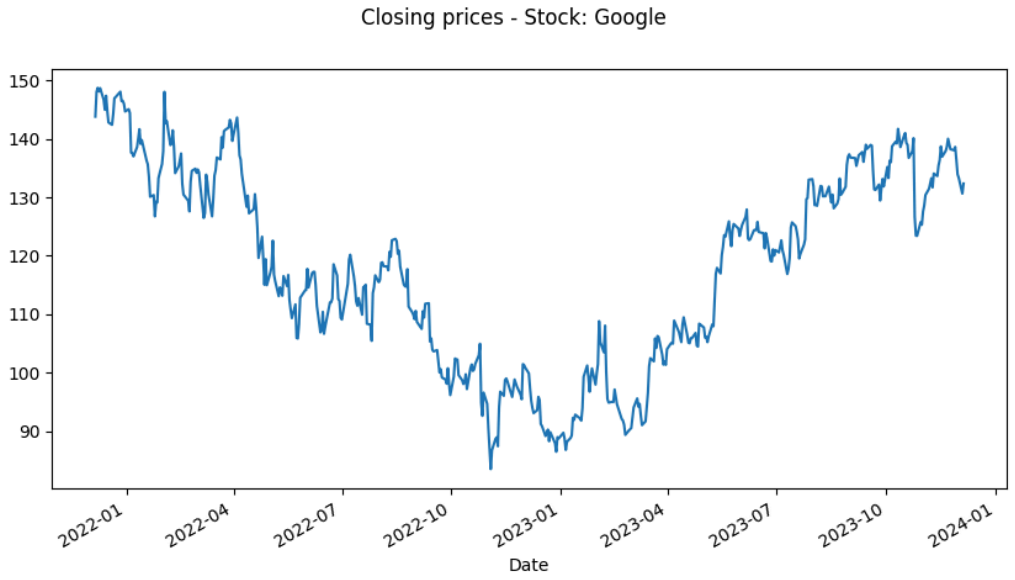


Figure 2.6. Closing Prices throughout time for Google

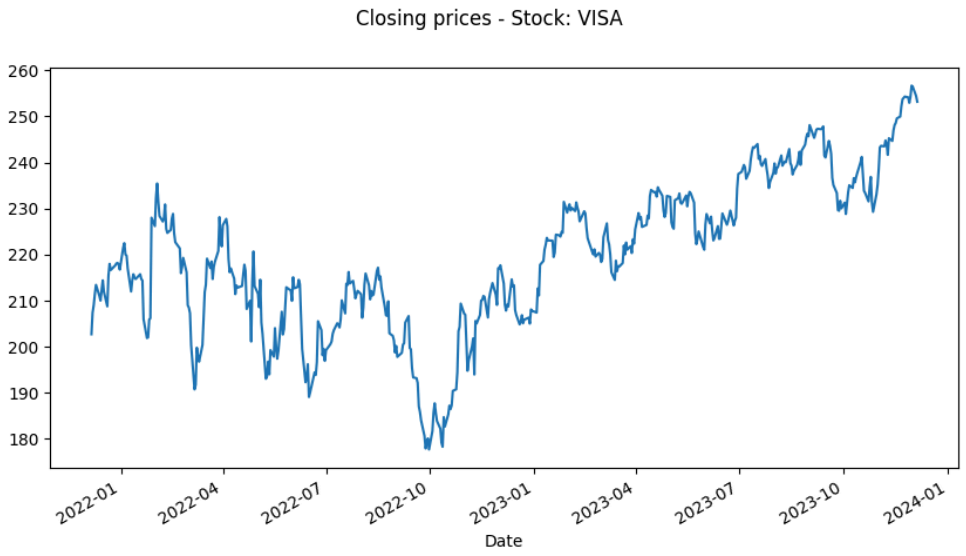


Figure 2.7. Closing Prices throughout time for Visa

2.3.2. Data Preprocessing

A split date is selected to divide the data in training and testing, in this case throughout all methodologies '2023-01-01' was used. For each company a training vector was created with all the closing prices from January 2022 to December 2023.

```
Python
traininig_data_msft = df_msft.loc[df_msft.index < split_date]
traininig_data_goog = df_goog.loc[df_goog.index < split_date]
training_data_visa = df_v.loc[df_v.index < split_date]
```

Next, data is normalized in order to train the model adequately.

A variable ‘prediction_days’ is set to 26, indicating the number of days to be considered for predicting stock prices. Then, the code initializes empty lists for training data input and output for three different stock symbols: Microsoft (msft), Google (goog), and Visa (v).

```
Python
prediction_days = 26

# Initialize empty lists for training data input and output
x_train_msft, y_train_msft = [], []
```

```
x_train_goog, y_train_goog = [], []
x_train_v, y_train_v = [], []
```

It iterates through the data, populating lists with previous ‘prediction_days’ values for each stock symbol and the corresponding current values. The lists are then converted to NumPy arrays and reshaped into 3D arrays, making them suitable for training Long Short-Term Memory (LSTM). This process enables the model to learn patterns from the past stock prices to predict future values for each stock symbol.

2.3.3. Model Training and Testing

Using the Keras library and Tensorflow, the ‘LSTM_model’ function is defined:

```python
def LSTM_model(x_train):
	# Initialize a sequential model
	model = Sequential()

	# Add the first LSTM layer with 50 units, input shape, and return sequences
	model.add(LSTM(units=75, return_sequences=True, input_shape=(x_train.shape[1], 1)))
	# Add dropout to prevent overfitting
	model.add(Dropout(0.2))

	# Add a second LSTM layer with 50 units and return sequences
	model.add(LSTM(units=75, return_sequences=True))
	# Add dropout to prevent overfitting
	model.add(Dropout(0.2))

	# Add a third LSTM layer with 50 units
	model.add(LSTM(units=75))
	# Add dropout to prevent overfitting
	model.add(Dropout(0.2))

	# Add a dense output layer with one unit
	model.add(Dense(units=1))

	return model
```

The LSTM model is composed of one input layer, two hidden layers and one output layer with 75 units in each layer. The parameter ‘units’ determines the dimensionality

of the output space of the LSTM layer. The Keras library –within the LSTM memory cell– applies the hyperbolic tangent activation function (TanH) and the Sigmoid activation function.

Next, the model is called for each company using between 25 to 45 epochs depending on the performance and accuracy of the model.

```python
model_msft = LSTM_model(x_train_msft)
model_msft.compile(optimizer='adam', loss='mean_squared_error')
model_msft.fit(x_train_msft, y_train_msft, epochs=35, batch_size=32,
callbacks=[checkpointer_msft])

model_goog = LSTM_model(x_train_goog)
model_goog.compile(optimizer='adam', loss='mean_squared_error')
model_goog.fit(x_train_goog, y_train_goog, epochs=25, batch_size=32,
callbacks=[checkpointer_goog])

model_v = LSTM_model(x_train_v)
model_v.compile(optimizer='adam', loss='mean_squared_error')
model_v.fit(x_train_v, y_train_v, epochs=45, batch_size=32,
callbacks=[checkpointer_v])
```

Three callbacks are employed to save the weights of the best-performing models based on the validation loss.

Then, the script processes test data for the three companies (Microsoft, Google, and Visa). The split date (January 2023 to January 2024) is used to create a testing vector for each company:

```python
test_data_msft = df_msft.loc[df_msft.index >= split_date]
test_data_goog = df_goog.loc[df_goog.index >= split_date]
test_data_v = df_v.loc[df_v.index >= split_date]
```

The training and test data are concatenated along the 'Close' column for each company. To generate predictions, the script initializes empty lists for test data input

and a loop iterates through the model inputs, creating sequences of data with the length of ‘prediction_days’.

The trained LSTM models (‘model_msft’, ‘model_goog’, and ‘model_v’) are then used to predict the future stock prices for the respective companies. The resulting variables hold the predicted closing stock prices based on the LSTM models trained on historical stock price data.

```python
predicted_prices_msft = model_msft.predict(x_test_msft)
predicted_prices_goog = model_goog.predict(x_test_goog)
predicted_prices_v = model_v.predict(x_test_v)
```

These predictions are then compared with the actual closing prices to evaluate the performance of the models on the test data.

2.3.4. Results

For each company evaluated, the actual and predicted price were compared along with three accuracy and performance indicators: the Mean Absolute Error (MAE), the Mean Absolute Percentage Error (MAPE), and the Mean Squared Error (MSE). MAE represents the average absolute difference between the predicted and actual values, giving a clear indication of how far off, on average, the predictions are from the actual stock prices. For stock price prediction –where the scale of prices can vary considerably between companies– MAPE offers a normalized measure of accuracy. In the context of stock prices –where extreme values or sudden changes can occur, depending on how volatile are the company stocks’– MSE is useful for penalizing larger errors more significantly, however, it is sensitive to outliers and amplifies the impact of prediction errors.

For the Microsoft share, the following is a graph that evaluates the predicted price of the stock using the LSTM Model vs. the actual price of the stock:

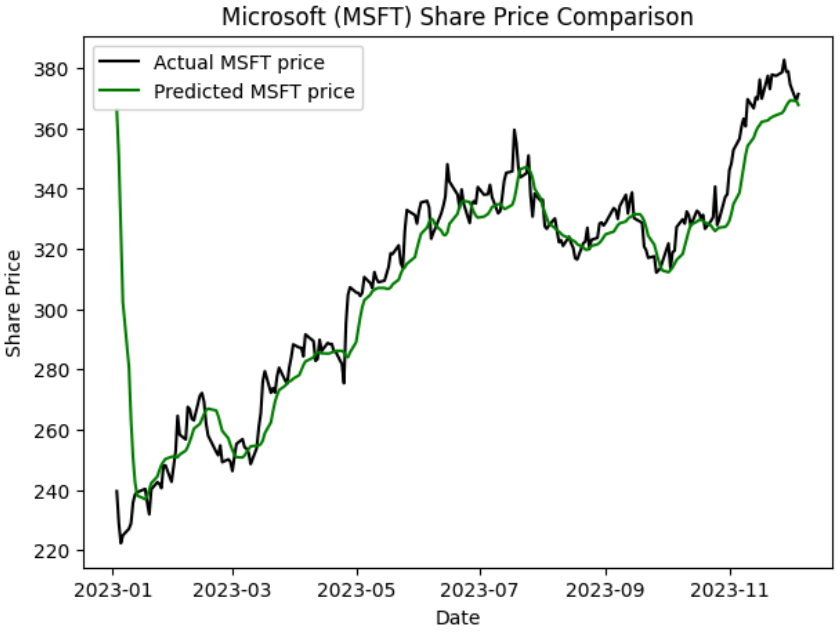


Figure 2.8. Predicted vs. Actual Prices for Microsoft

Actual price 06-12-2023	**Prediction 06-12-2023**	**MAE**	**MAPE**	**MSE**
368.80	365.59	8.74	13.94%	290.94

Table 2.1. Results and Metrics for Microsoft

For the Google share, the following is a graph that evaluates the predicted price of the stock using the LSTM Model vs. the actual price of the stock:

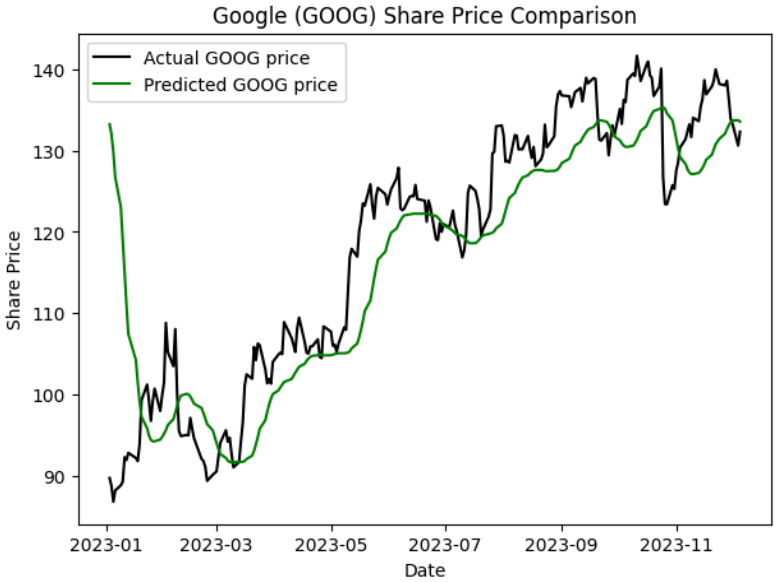


Figure 2.9. Predicted vs. Actual Prices for Google

Actual price 06-12-2023	Prediction 06-12-2023	MAE	MAPE	MSE
131.43	133.29	6.09	15.24%	79.91

Table 2.2. Results and Metrics for Google

For the Google share, the following is a graph that evaluates the predicted price of the stock using the LSTM Model vs. the actual price of the stock:

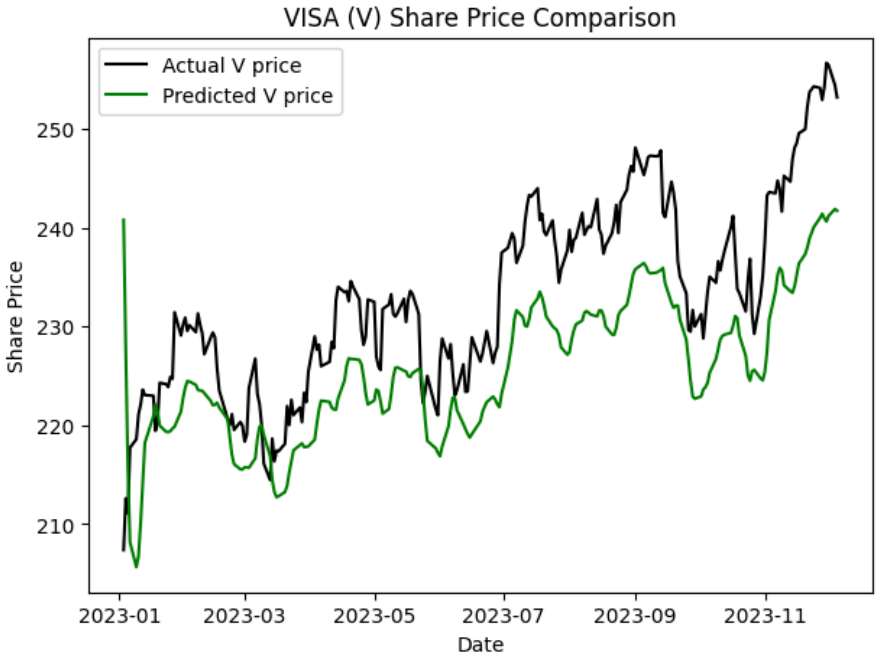


Figure 2.10. Predicted vs. Actual Prices for Visa

Actual price 06-12-2023	Prediction 06-12-2023	MAE	MAPE	MSE
254.29	240.8	7.81	4.73%	76.44

Table 2.3. Results and Metrics for Visa

As seen throughout the **figures 2.8 to 2.10.** there's a spike at the beginning of the predicted price curve, this spike causes the high values observed in the MSE metric. The reason for this unusual spike (considering the accuracy of the behavior through the rest of the curve) is unknown but one explanation may be that during the initial stages of prediction, the model might take some time to adjust to the patterns and features present in the data, furthermore the sudden spike could be a result of the

model's initial learning and adaptation to the training data. As a future improvement, the model would need more data to train on since in this case the amount of data was limited (roughly one year worth of information); adding data would increase the complexity of the neural network but by fine-tuning some of the model's hyperparameters it would improve the the overall performance and accuracy of the model.

2.4. Stock Market Simulation

Setting a balance of ten thousand dollars for each stock, the stock market simulation is implemented to observe how the predictions made by the LSTM model perform in a trading environment. As expected, the overall gains over time are greater within a bigger time period.

Days	GOOGLE	MSFT	VISA
7	$10,000.00	$10,000.00	$10,045.68
30	$9,428.06	$11,291.74	$10,201.08
60	$10,556.20	$11,569.97	$9,785.51
90	$12,184.31	$11,789.54	$9,657.76

Table 2.4. Results of the Stock Market Simulation

3. Random Forest

Decision trees and random forests are widely used machine learning techniques for solving classification or regression problems. Decision trees operate by recursively dividing the feature space using a tree structure, with each division continuing until pure nodes are reached. Pure nodes contain samples of only one class and are the ultimate leaf nodes of the tree. During classification, a test sample traverses the tree to a leaf node, where it inherits the class label of the samples in that node. Random forests employ an ensemble of decision trees to mitigate overfitting. Each tree in a random forest is built on a subset of the feature space, defined by the following formula:

$$m = \sqrt{M} \tag{3. 1}$$

where:

- *m = subset size*
- *M = total number of features*

3.1. Understanding Decision Trees

Decision trees represent a fundamental concept in machine learning. These trees are constructed through recursive partitioning of the feature space (the set of all possible combinations of features or variables that can be used to describe or represent data), forming a hierarchical structure where each node represents a decision point based on a particular feature. As the tree branches out, it progressively divides the feature space into smaller regions, aiming to maximize purity in each resulting subset. There are two types of decision trees, classification and regression, however this paper will focus only on the classification type. In Random Forest decision tree learners are built by randomly choosing m out of M features and n out of N samples; the training and test sets are mutually exclusive.

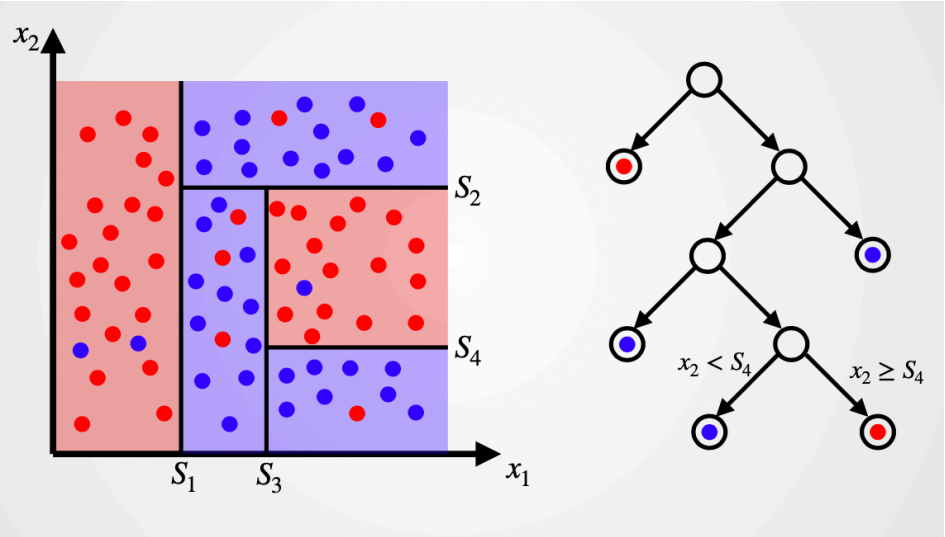


Figure 3.1 Visualization of decision trees creation[5]

The primary objective of classification decision trees is to categorize data points by iteratively splitting them into subsets that contain instances of a single class, ultimately leading to leaf nodes representing pure classes. During the training process, the algorithm selects the most informative feature and its corresponding threshold to split the data at each node. This decision-making process is guided by criteria such as Gini impurity and Shannon Entropy, which measure the homogeneity or purity of the data within each partition.

Gini impurity assesses the likelihood of misclassifying a randomly chosen element if it were labeled according to the distribution of classes in the node. Conversely, Shannon Entropy quantifies the uncertainty or disorder in the data, with higher entropy indicating greater unpredictability.

Gini impurity at node N is given by:

$$G(N) = 1 - (P_1)^2 - (P_{-1})^2 \tag{3.2}$$

where:

[5] Khowaja, K., & Härdle, W. K. (November 2023). Trespassing Random Forests: Exploring Random Forests and Relevant Work at IRTG 1792. Retrieved from: https://quantinar.com/course/246/trespassing-random-forests

- *G(N) = impurity in node N*
- *Pi = proportion of the population with class label i*

Entropy at node N is given by:

$$S(N) = - P_1 log(P_1) - P_{-1} log(P_{-1})$$

(3.3)

where:

S(N): entropy in node N
d: number of classes considered and P(ωi) is the proportion of the population labeled as i.

At its core, decision tree construction (best split), involves selecting the best attribute that maximizes the information gain or minimizes the impurity, calculated as follows:

$$Gain = I(N) - p_L I(N_L) - p_R I(N_R)$$

(3.4)

where:

Gain: Information gain
- *I(N) = impurity measure of node N*
- *I(NL) = impurity in the left child node of N after the split*
- *I(NR) = impurity in the right child node of N after the split*
- *PL = proportion of the same in the left children node*
- *PR = proportions of the samples in the left and right children nodes*

Subsequently, at the next level of the tree, each child node undergoes a similar process, but this time, the partitions of the training sample space belonging to the children nodes are considered. This recursive partitioning continues until pure nodes are reached and **all leaf nodes become pure** (cannot be further split)**.** In decision trees, all leaf nodes must be pure nodes, which means that they are not split further.

A node is considered pure if all the samples in that node belong to the same class. When an element needs classification, it traverses through the tree based on partitioning criteria until it reaches a leaf node, where it inherits the class label of all the elements in that leaf node. The range goes from 0 if all elements belong to the same class and 1 if only one class exists.

By utilizing thresholds determined via Gain for each split, the feature space can be partitioned into pockets, where each partition corresponds to a specific class. Figure 2.1 illustrates the partitions of the feature space for Tree 1, with blue partitions representing Class 1 and red partitions representing Class 2. This underscores how random forests effectively handle non-linear data trends or separability issues.

3.2. Understanding Random Forest

The primary issue with using only decision trees for classification tasks is their tendency to overfit the training data. Overfitting occurs when a model learns the training data too well, capturing noise and random fluctuations instead of underlying patterns. As decision trees recursively partition the feature space to achieve purity in leaf nodes, they may create excessively complex structures that fit the training data perfectly but generalize poorly to unseen data. This results in high variance, where small changes in the training data can lead to significantly different tree structures and predictions.

Moreover, decision trees can be sensitive to the specific characteristics and distribution of the training data. If the training data is imbalanced or contains outliers, decision trees may produce biased or inaccurate predictions. Additionally, decision trees are prone to being influenced by irrelevant features, leading to suboptimal splits and reduced predictive performance.

Bootstrap aggregating, also known as bagging, is employed as an ensemble learning method to enhance the stability and accuracy of learning algorithms, mitigating issues such as variance and overfitting commonly encountered in decision tree construction.

Bagging generates multiple new datasets by sampling uniformly from the original dataset with replacement. Sampling with replacement involves randomly selecting instances from a dataset with the possibility of selecting the same instance multiple times. This implies that the probability of a feature being successfully chosen to grow a decision tree is:

$$P(k\ successes) = \binom{n}{k} p^{k} q^{n-k} \tag{3.5}$$

where:

probability that a feature xa is used in growing k out of n trees

- *p = probability of success*
- *q = probability of failure*

This process enables the creation of new datasets with the same size as the original but potentially containing duplicate instances. Bagging offers certain advantages, particularly in high-dimensional datasets, where each feature's probability of being chosen to grow a decision tree is binomially distributed based on the number of trees being constructed (Basak, 2023). It also ensures that with the increase in the number of tree estimators in a random forest, the chance of error decreases, adding stability to the process since the forest as a whole is representative of the class to which a sample to be classified belongs.

Aggregation refers to the process of combining the predictions of multiple models into a single prediction by averaging. After training several individual models on different subsets of the original dataset, each model produces its own set of predictions. Aggregation involves synthesizing these individual predictions to form a final prediction that is more accurate and robust than any single model's prediction.

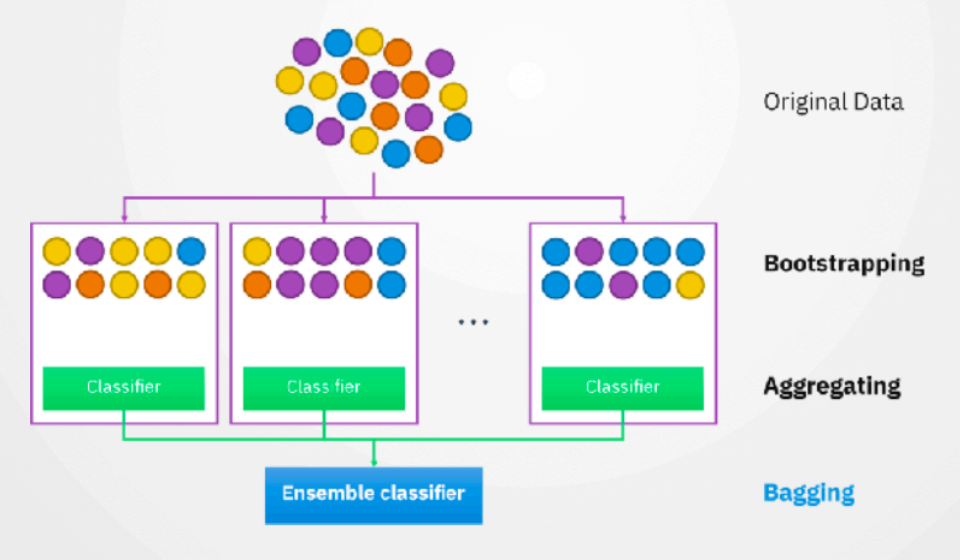


Figure 3.2. Visualization on how bagging applied to Random Forest works [6]

```
Python

Algorithm 1: Decision Tree

input: X = (xi,yi) is the labeled training data
l is the current level of the tree
M is the set of features used to grow a tree

output: A tree which is configured to predict the class label of a test sample

l ← 1+1;
CL ← null;
CR ←null;
; /* CL, CR are the left and right children of this node respectively */
MaxGain ← 0;
; /* MaxGain stores the value of the maximum possible gain that can be achieved from
splitting a node based on this, the final split is determined */
for j in M do
       Sort (xi, yi)^ n in the increasing order of the jth feature
       for ← 1 to n do
              CL ← X[0:i];
              CR ← X[i+1:n];
              Gain ← GiniGain ((xi, yi)^n[j], CL, CR);
              if Gain > MaxGain then
                     MaxGain ← Gain;
                     CL, CR ← CL, CR;
       end
```

[6] Basak, S., Kar, S., Saha, S., Khaidem, L., & Roy Dey, S. (2019). Predicting the direction of stock market prices using tree-based classifiers. The North American Journal of Economics and Finance, 47, 552-567. https://doi.org/10.1016/j.najef.2018.06.013

```
end

if CL does not satisfy the desired level of purity then
       DecisionTree(CL, l, M);
if CR does not satisfy the desired level of purity then
       DecisionTree(CR, L, M);
; /* A node is not pure when it has samples belonging to more than one class. In such a
case, it needs to be split in turn. In this way, through successive function calls,
the feature space is recursively partitioned. */
```

Figure 3.3. Decision Tree on Random Forest Algorithm[7]

3.3. OOB error and Convergence of the Random Forest

The out-of-bag error is a method used to estimate the generalization error of a model. During the training process, each decision tree is built using a bootstrap sample of the original dataset. The out-of-bag examples are the data points that are not included in the bootstrap sample used to train a particular decision tree. Because these data points were not used to build the tree, they can be used to evaluate the performance of the tree. The out-of-bag error is then calculated as the average error across all out-of-bag examples for all trees in the forest. This provides an estimate of how well the model will perform on unseen data.

When we construct the ensemble of decision trees, we're essentially creating multiple models, each trained on a different subset of the original training data. For each training sample Zi=(Xi,Yi) within the original training set T, we find all the bagged sets Tk that do not contain Zi. For every n data samples in the original training dataset, there are n such sets available. We compute the out-of-bag (OOB) error by averaging the errors for each Zi, using predictions from the trees that did not include Zi in their respective bootstrap sample.

[7] Basak, S., Kar, S., Saha, S., Khaidem, L., & Roy Dey, S. (2019). Predicting the direction of stock market prices using tree-based classifiers. The North American Journal of Economics and Finance, 47, 552-567. https://doi.org/10.1016/j.najef.2018.06.013

Convergence refers to the behavior of a model or algorithm as it iteratively approaches a stable or optimal solution. For example, in the context of training a model using an iterative optimization algorithm such as gradient descent, convergence occurs when the algorithm's updates to the model parameters become progressively smaller and the model's performance stabilizes. This refers to the behavior of the model's error rate as more trees are added to the ensemble. Ideally, as the number of trees increases, the error rate should decrease and eventually stabilize. This indicates that adding more trees to the forest is no longer improving the model's performance, and the model has converged to its optimal state given the training data and parameters. This observation also elucidates why random forests are resistant to overfitting as the ensemble grows.

The ensemble model's prediction at each stage is an accumulation of the predictions made by all the base learners up to that point, with each base learner's contribution scaled by a weight (γm) that reflects its importance or relevance in the ensemble. This iterative process continues until the desired number of stages is reached or until some convergence criterion is met. Defined by:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{3.6}$$

where:

m = current stage or iteration.

hm(x) =current base learner.

Fm(x) = output of the ensemble model at the m-th stage.

Fm−1(x) = output of the ensemble model at the previous stage (m-1).

γm = weight assigned to the prediction by hm(x) at the m-th stage.

hm(x) = prediction made by hm(x) at the m-th stage.

3.4. Evaluation

Various metrics are employed to evaluate the performance of classification models. These metrics provide insights into different aspects of a model's effectiveness in correctly predicting class labels. Here, we'll delve into four fundamental evaluation metrics: accuracy, recall, precision, and specificity. Additionally, we'll introduce the F1 score, a metric that combines precision and recall to provide a balanced measure of a model's performance.

- Accuracy (Acc):

Measures the overall correctness of predictions made by a classification model. It is calculated as the ratio of the number of correctly classified instances to the total number of instances:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{3.7}$$

- Recall (Sensitivity, True Positive Rate):

Also known as sensitivity or true positive rate, quantifies the model's ability to correctly identify positive instances out of all actual positive instances. It is calculated as the ratio of true positive predictions to the total number of actual positive instances:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3.8}$$

- Precision:

Measures the proportion of correctly identified positive instances out of all instances predicted as positive by the model. It is calculated as the ratio of true positive predictions to the total number of instances predicted as positive:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3.9}$$

- Specificity (True Negative Rate):

Also known as the true negative rate, measures the model's ability to correctly identify negative instances out of all actual negative instances. It is calculated as the ratio of true negative predictions to the total number of actual negative instances:

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{3.10}$$

- F1 Score:

The F1 score is the harmonic mean of precision and recall, providing a balanced measure of a model's performance. It is calculated as:

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.11}$$

This metric is particularly useful when the class distribution is imbalanced, as it considers both false positives and false negatives.

3.5. Optimization

Hyperparameters are parameters that are set before the learning process begins. They are not directly learned from the data but rather are chosen beforehand and

remain constant throughout the training process. Examples of hyperparameters include the learning rate of an algorithm or the depth of a decision tree.

To improve the stock market prediction we implemented hyperparameter tuning, particularly in the context of model optimization. It is often employed in conjunction with models that have a large number of hyperparameters or when exhaustive search methods become computationally expensive.

The primary purpose is to identify the optimal set of hyperparameters for a given machine learning model. It systematically explores a predefined range of hyperparameter values and selects the combination that yields the best performance according to a specified evaluation metric, carried out using cross-validation.

Cross-validation is a technique used to assess the performance of a model by splitting the data into multiple subsets, training the model on some subsets, and evaluating it on the remaining subsets. This helps to ensure that the model's performance is robust and not overly sensitive to the particular way in which the data is divided.

3.6. Results

Based on the results obtained from the Random Forest algorithm we have several outputs:

1. Confusion matrix: This matrix shows the performance of the model in terms of true positives, false positives, true negatives, and false negatives. It helps in assessing the accuracy, precision, recall, and F1-score of the model's predictions.
2. ROC curve: The receiver operating characteristic (ROC) curve illustrates the trade-off between the true positive rate (sensitivity) and the false positive rate (1 - specificity) across different threshold values. It helps in evaluating the model's ability to discriminate between classes.
3. Feature Importance graphs: These graphs display the importance of each feature (or predictor variable) in the model. They help in understanding which features are most influential in predicting the direction of stock prices.

- Google

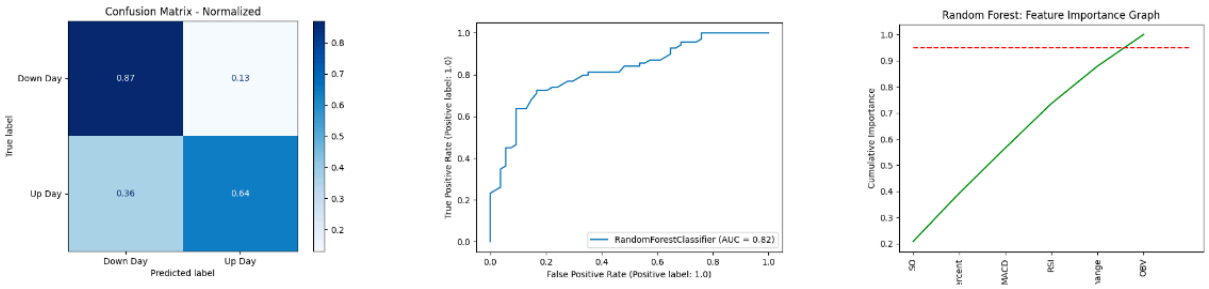


Figure 3.5. Confusion matrix, ROC curve and Feature Importance graphs before optimization for Google stock

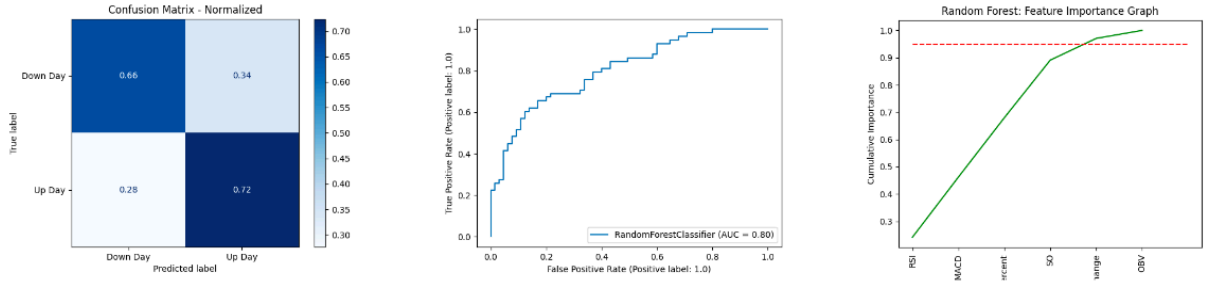


Figure 3.6. Confusion matrix, ROC curve and Feature Importance graphs after optimization for Google stock

Correct Prediction (%): 73.98373983739837

Classification Report:	precision	recall	f1-score	support
Down Day	0.652778	0.870370	0.746032	54.000000
Up Day	0.862745	0.637681	0.733333	69.000000
accuracy	0.739837	0.739837	0.739837	0.739837
macro avg	0.757761	0.754026	0.739683	123.000000
weighted avg	0.770564	0.739837	0.738908	123.000000

Accuracy:	0.7398373983739838
Percision:	0.8627450980392157
Recall:	0.6376811594202898
Specificity:	0.8703703703703703

SO	0.207610
R_percent	0.184823
MACD	0.173177
RSI	0.170134
Price_Rate_Of_Change	0.143105
OBV	0.121151

Random Forest Out-Of-Bag Error Score: 0.7029972752043597

Correct Prediction (%): 69.10569105691057

Classification Report:	precision	recall	f1-score	support
Down Day	0.728814	0.661538	0.693548	65.000000
Up Day	0.656250	0.724138	0.688525	58.000000
accuracy	0.691057	0.691057	0.691057	0.691057
macro avg	0.692532	0.692838	0.691036	123.000000
weighted avg	0.694597	0.691057	0.691179	123.000000

Accuracy:	0.6910569105691057
Percision:	0.65625
Recall:	0.7241379310344828
Specificity:	0.6615384615384615

RSI	0.241608
MACD	0.220401
R_percent	0.217589
SO	0.211343
Price_Rate_Of_Change	0.079651
OBV	0.029409

Random Forest Out-Of-Bag Error Score: 0.7158469945355191

Figure 3.7. Detailed metrics before and after optimization for Google stock respectively.

- MSFT

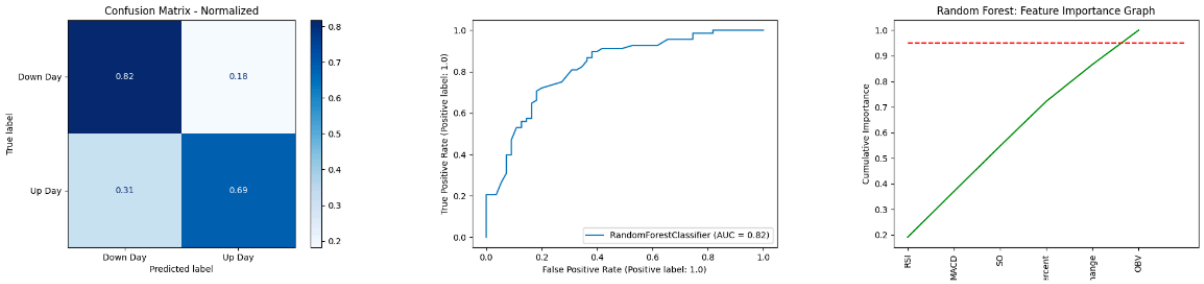


Figure 3.8. Confusion matrix, ROC curve and Feature Importance graphs before optimization for Microsoft stock.

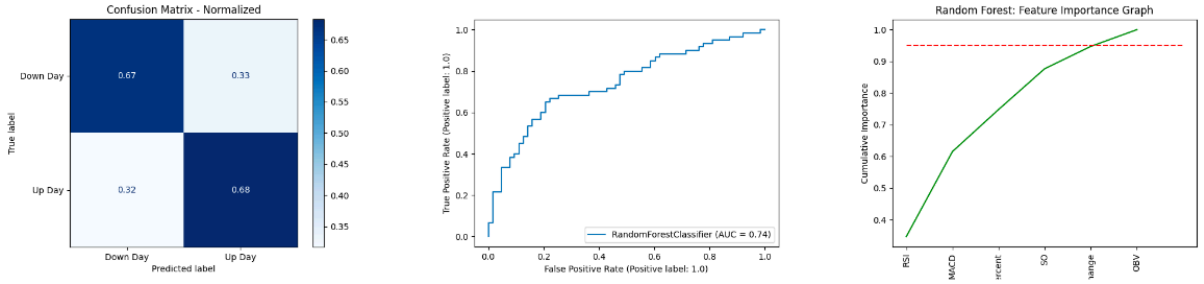
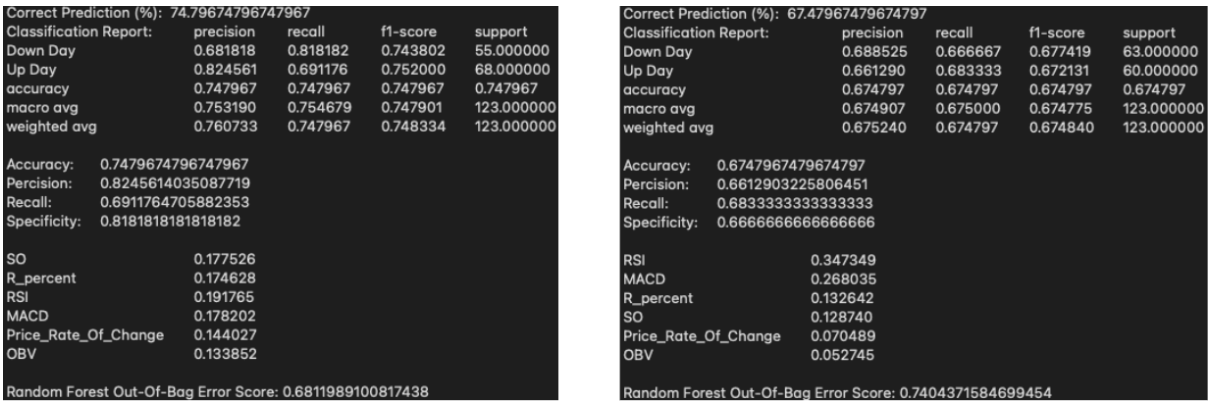


Figure 3.9. Confusion matrix, ROC curve and Feature Importance graphs after optimization for Microsoft stock.



Correct Prediction (%): 74.79674796747967

Classification Report:	precision	recall	f1-score	support
Down Day	0.681818	0.818182	0.743802	55.000000
Up Day	0.824561	0.691176	0.752000	68.000000
accuracy	0.747967	0.747967	0.747967	0.747967
macro avg	0.753190	0.754679	0.747901	123.000000
weighted avg	0.760733	0.747967	0.748334	123.000000

Accuracy: 0.7479674796747967
Percision: 0.8245614035087719
Recall: 0.6911764705882353
Specificity: 0.8181818181818182

SO	0.177526
R_percent	0.174628
RSI	0.191765
MACD	0.178202
Price_Rate_Of_Change	0.144027
OBV	0.133852

Random Forest Out-Of-Bag Error Score: 0.6811989100817438

Correct Prediction (%): 67.47967479674797

Classification Report:	precision	recall	f1-score	support
Down Day	0.688525	0.666667	0.677419	63.000000
Up Day	0.661290	0.683333	0.672131	60.000000
accuracy	0.674797	0.674797	0.674797	0.674797
macro avg	0.674907	0.675000	0.674775	123.000000
weighted avg	0.675240	0.674797	0.674840	123.000000

Accuracy: 0.6747967479674797
Percision: 0.6612903225806451
Recall: 0.6833333333333333
Specificity: 0.6666666666666666

RSI	0.347349
MACD	0.268035
R_percent	0.132642
SO	0.128740
Price_Rate_Of_Change	0.070489
OBV	0.052745

Random Forest Out-Of-Bag Error Score: 0.7404371584699454

Figure 3.10. Detailed metrics before and after optimization for Microsoft stock respectively.

- **V**

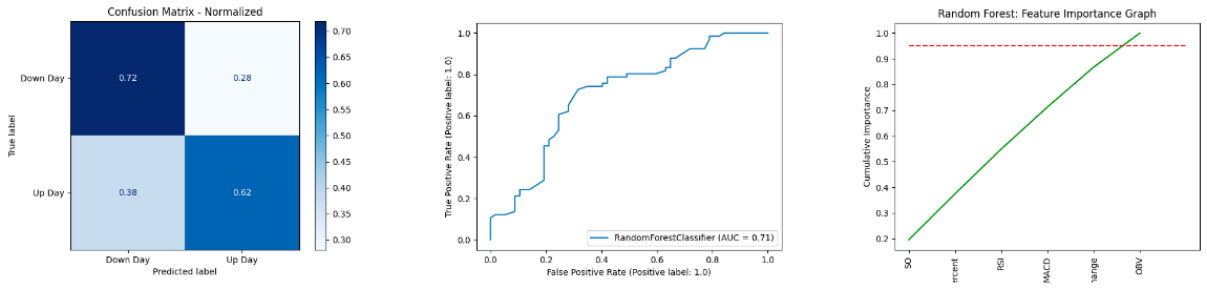


Figure 3.11. Confusion matrix, ROC curve and Feature Importance graphs before optimization for Visa stock

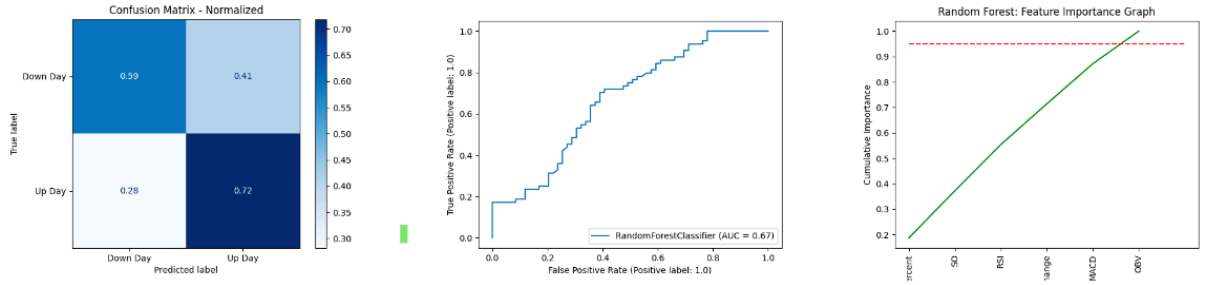
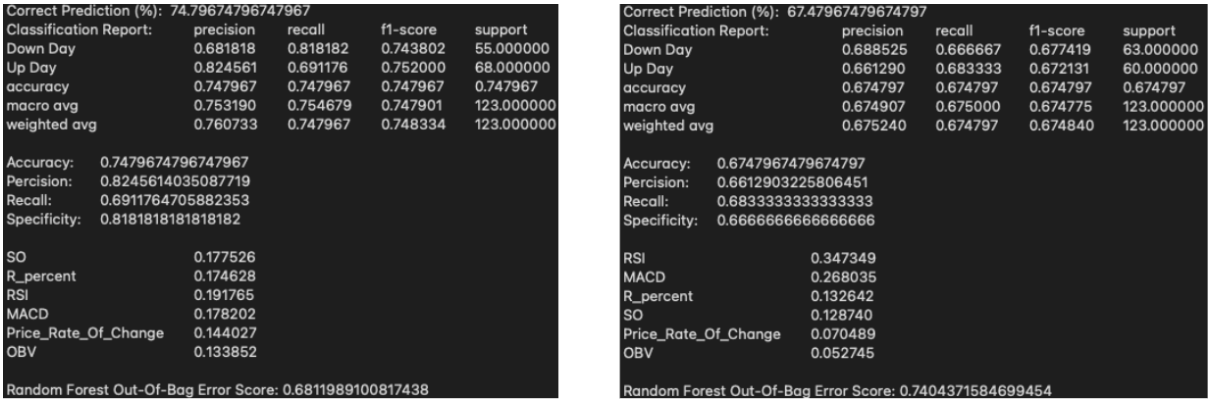


Figure 3.12. Confusion matrix, ROC curve and Feature Importance graphs after optimization for Visa stock



Correct Prediction (%): 74.79674796747967

Classification Report:	precision	recall	f1-score	support
Down Day	0.681818	0.818182	0.743802	55.000000
Up Day	0.824561	0.691176	0.752000	68.000000
accuracy	0.747967	0.747967	0.747967	0.747967
macro avg	0.753190	0.754679	0.747901	123.000000
weighted avg	0.760733	0.747967	0.748334	123.000000

Accuracy: 0.7479674796747967
Percision: 0.8245614035087719
Recall: 0.6911764705882353
Specificity: 0.8181818181818182

SO	0.177526
R_percent	0.174628
RSI	0.191765
MACD	0.178202
Price_Rate_Of_Change	0.144027
OBV	0.133852

Random Forest Out-Of-Bag Error Score: 0.6811989100817438

Correct Prediction (%): 67.47967479674797

Classification Report:	precision	recall	f1-score	support
Down Day	0.688525	0.666667	0.677419	63.000000
Up Day	0.661290	0.683333	0.672131	60.000000
accuracy	0.674797	0.674797	0.674797	0.674797
macro avg	0.674907	0.675000	0.674775	123.000000
weighted avg	0.675240	0.674797	0.674840	123.000000

Accuracy: 0.6747967479674797
Percision: 0.6612903225806451
Recall: 0.6833333333333333
Specificity: 0.6666666666666666

RSI	0.347349
MACD	0.268035
R_percent	0.132642
SO	0.128740
Price_Rate_Of_Change	0.070489
OBV	0.052745

Random Forest Out-Of-Bag Error Score: 0.7404371584699454

Figure 3.13. Detailed metrics before and after optimization for Visa stock respectively.

3.7. Stock market simulation

Metrics	GOOG	MSFT	V
Correct Prediction	69.10	67.47	65.85
Accuracy	0.69	0.67	0.65
Out-Of-Bag	0.71	0.74	0.66

Table 3.1. Summary of relevant metrics

Days	GOOG	MSFT	V
7	$7,504.29	$9,088.34	$7,124.58
30	$20,428.04	$19,963.95	$9,032.84
60	$43,432.95	$11,934.86	$4,585.14
90	$33,186.53	$3,837.84	$3,091.25

Table 3.2. Return of investments before optimization

Days	GOOG	MSFT	V
7	$7,504.29	$7,185.86	$7,411.16
30	$20,608.03	$15,023.71	$7,506.98
60	$59,505.92	$9,204.86	$3,845.99
90	$67,393.19	$4,193.66	$3,059.73

Table 3.3. Return of investments after optimization

The analysis of the stock market simulation data provides a comprehensive overview of the predictive performance and investment profitability across Google (GOOG), Microsoft (MSFT), and Visa (V).

Beginning with the summary of relevant metrics in Table 3.1, it is evident that the model demonstrates relatively high correct prediction rates, with GOOG leading at 69.10%, followed closely by MSFT at 67.47%, and V at 65.85%. The corresponding accuracy metrics, ranging from 0.65 to 0.69, indicate the proportion of correct predictions out of the total predictions made. Moreover, the out-of-bag estimates, ranging from 0.66 to 0.74, offer insights into the prediction error, with lower values suggesting higher prediction accuracy.

Moving to the return of investments (ROI) data presented in Tables 3.2 and 3.3, several key observations emerge. Before optimization, GOOG exhibits varying ROI figures across different time intervals, ranging from $7,504.29 to $67,393.19, indicating the potential profitability of investments over short to medium-term periods. MSFT and V also showcase fluctuating ROI values, with MSFT demonstrating significant returns particularly over longer time horizons.

After optimization, there is a noticeable improvement in ROI figures for all stocks, highlighting the positive impact of model refinement. For instance, GOOG's ROI increases across all time intervals, suggesting enhanced profitability post-optimization. Similarly, MSFT and V also experience improvements in ROI to varying degrees.

4. XGBoost

4.1 What is XGBoost?

XGBoost is a gradient boosting library that implements machine learning algorithms using the gradient boosting technique. This in order to provide tree boosting functions for solving machine learning and data science problems more efficiently than traditional neural networks and machine learning algorithms. It provides parallel tree boosting for regression, ranking and classification problems, this last one is what concerns us in this report since we aim to determine whether there is an up or down tendency for the stock prices.

The XGBoost random forest classifier that we will be using later on is a supervised machine learning method. This means that it uses algorithms to train the model using a labeled dataset, where the models learns about the information in order to find patterns in the data provided by the features and labels, and then uses the trained model to predict the labels on a new dataset with the same features or for the testing split as in **figure 3.1**.

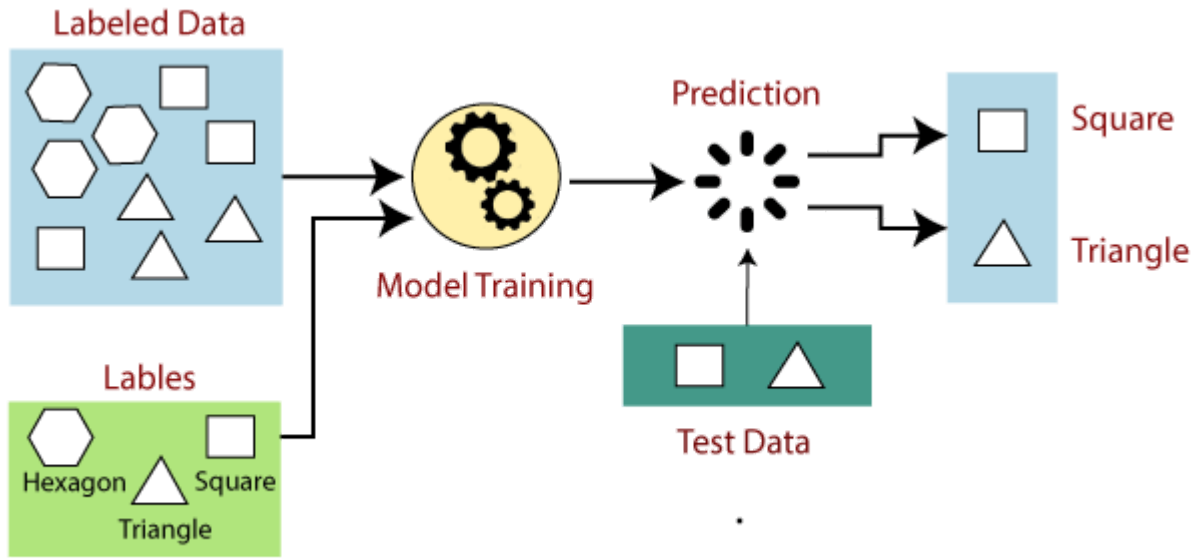


Figure 4.1: Supervised ML model

There are two types of supervised machine learning algorithms. The first is regression, which we use in scenarios where our input and output variables are related and only if the output variables are continuous variables. For our implementation, we will use the second type, which is classification. This algorithm works for problems in which the output variable will be categorical. This is the case since our output variable is a class of the tendency to go up or down in price.

XGboost uses gradient boosted decision trees for both classification and regression algorithms, these decision trees are ensemble learning algorithms that work similarly to random forest. By ensemble learning algorithm, we mean that the algorithms use a technique that enhances their accuracy by using multiple models in order to reduce any error or bias that may be present in the individual models. This technique combines the outputs of the individual models and creates a more precise prediction.

As the name suggests, XGBoost uses a boosting technique for its ensemble learning model, but it implements a method called extreme Gradient Boosting. In order to understand how XGBost works with this technique, we need to understand how regular boosting works. It is a sequential process in which the current model iteration tries to fix the errors of the previous model, so the models depend on each other for this to work. The boosting technique follows a series of steps:

1. Create an initial subset from the dataset with all data points having equal weights
2. Create a base model
3. Make predictions on the whole data set
4. Calculate the errors using predicted and real values
5. The errors are given higher weights for the next iteration
6. Create a new model to make new predictions on the dataset, but aiming to correct the previous model's errors.
7. Do the same with multiple models
8. In the end, the final model is the weighted average of all the models, as in **figure 4.2**

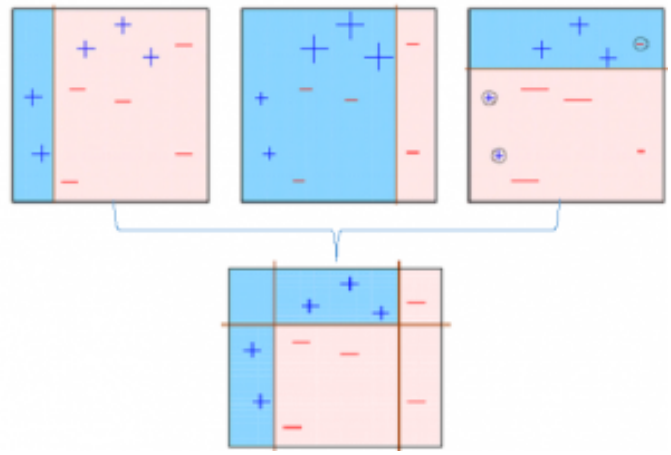


Figure 4.2: Strong learner formed by weak learners

Having as a result a strong learner formed by the weighted average of the weak learners that didn't perform as good as intended on the dataset nut, they boosted the performance of the resulting model.

4.2 What is gradient boosting?

As the name states, it is a boosting algorithm used for classification and regression problems. As we mentioned previously, this kind of ensemble learning methods train models sequentially to correct errors and combine the weak learners to form strong learners.

Another boosting method called Ada Boost the weights of incorrect classifications are increased for the next model to focus on these samples, but using gradient boosting does not adjust the weights because the next iteration uses the residual errors of the previous one as labels. This is done by calculating the negative gradient of the loss function in respect to the prediction output. The math behind the algorithm works as follows:

1. Assuming **X** as the input dataset, **Y** as the prediction target and **N** as the number of samples, we need to find the function **f(x)** that is able to map the features in **X** to the target **Y**. Loss function representing the difference between actual and predicted values

$$L(f) = \sum_{i=1}^{N} L(y_i, F(x_i))$$

2. Minimize loss of function with respect to **f**

$$f_0(x) = argmin_f L(f) = argmin_f \sum_{i=1}^{N} L(y_i, F(x_i))$$

Our gradient boosting algorithm has M iterations, so to improve the f_m algorithm we add a new estimator h_m having $1 \leq m \leq M$

$$y_i = F_{m+1}(x_i) + h_m(x_i)$$

3. For M iterations, the steepest descent finds $h_m = -\rho_m g_m$ where ρ_m is a constant known as step length and g_m is the gradient of the loss function.

$$g_{im} = \left[\frac{L(y_i, f(x_i))}{f(x_i)}\right]_{f(x_i) = f_{m-1}(x_i)}$$

4. The gradient similarly for M trees

$$f_m(x) = f_{m-1}(x) + \left(argmin_{h_m \in H}\left[\sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + h(x_i))\right]\right)(x)$$

The solution will be

$$f_m = f_{m-1} - \rho_m g_m$$

Extreme gradient boosting is an implementation of this algorithm that has proven to be highly effective and also faster than other gradient boosting techniques. Including regularization to improve performance and reduce overfitting. XGBoost implements parallel processing, is highly flexible since it allows custom optimization objectives and evaluation criteria, handles missing values on its own and creates splits until the model reaches the max depth defined for the trees, moves backwards and removes splits from trees if they don't have positive gains.

4.3 Why use XGBoost?

XGBoost has proven to be highly effective for solving regression and classification problems and has a wide amount of integrations with more tools in python and R for user to experiment with. It has a wide range of data scientist contributing to its development, also has proven to be effective in ranking and prediction problems. It is a highly portable tool and supports cloud integrations.

On the other hand, there are other techniques such as Ada Boosting, which stands for adaptive boosting. This is one of the most usual techniques getting compared with gradient boosting implementations such as XGBoost due to its similarity, but it has some clear differences that place gradient boosting as a better option in most cases. Some key differences are the following:

- Weight adjustment
    - Ada boost increased the weights for the incorrectly classified samples
    - Gradient boosting updates the weight by calculating the negative gradient of the loss function mentioned earlier in this paper with respect to the prediction
- Base learners
    - Ada boosting uses decision trees
    - Gradient boosting implements a wife variety of base learners such as decision trees and other linear models
- Reliability
    - AdaBoost tends to have more noise and outliers in the data due to the assignation of higher weights to misclassified samples
    - Gradient boosting is more robust due to the calculation of the gradient being less sensitive to outliers.

Overall, by the already exposed characteristics and comparison to other boosting methods, XGboost has a great combination of performance and processing time, which makes it a great option for the implementation in this paper.

4.4 Math behind XGBoost

XGBoost is based on function approximation by using a loss function and a regularization factor. The objective function at an iteration t that we have to minimize is as follows:

$$L^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

- "y" represents the real value from the training dataset
- $\Omega(f_t)$ is the regularization factor
- f_t function that maps the features to the target in the current iteration

$$\text{seen as } f(x + \Delta x) \text{ where } x = \hat{y}_i^{(t-1)}$$

This objective function cannot be optimized using traditional optimization. To transform the objective function to a function in the Euclidean domain, we need to use the Taylor approximation, since it allows us to use traditional optimization techniques once it is applied.

In Taylor approximation, the linear approximation of a function is the following:

$$f(x) \approx f(a) + f'(a)(x - a)$$

initial functions can be written as function of Δx only

$$\Delta x = f_t(x_i) = (x - a)$$

We can transform a function $f(x)$ to the simplest function Δx on a specific point a. The x in the objective function $f(x)$ represents the sum of t number of CART trees, becoming a function of the current step.

In this case $f(x)$ represents the loss function l, a is the previous step predicted value and Δx is the new learner that we will create. We can write the objective function as a simple function of the new learner to apply Euclidean space optimization techniques.

As we said a is the prediction at the step t -1 and $x - a$ represents our new learner for time t, to minimize the objective we apply the second order Taylor approximation.

$$f(x) \approx f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2$$

$$L^{(t)} \simeq \sum_{i=1}^{n}\left[l(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t)$$

Updated objective using second order Taylor approximation

Where

$$g_i = \partial_{y'^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \text{ and } h_i = \partial^2_{y'^{(t-1)}} l(y_i, y'^{(t-1)})$$

These two represent the first and second order gradients of the loss function. By removing the constant parts, we get the following simplified function to minimize at iteration t.

$$L^{(t)} \simeq \sum_{i=1}^{n}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t)$$

The simplified objective represents a sum of quadratic function that can be simplified.

$$argmin_x G_x + \frac{1}{2}Hx^2 = -\frac{G}{H},\ H > 0 \quad min_x Gx + \frac{1}{2}Hx^2 = -\frac{1}{2}\frac{G}{H}$$

Minimizing a quadratic function

The next step is to find a learner for iteration t that is able to reduce loss as much as possible. To measure the quality of a tree structure, “q” and the scoring function, we’ll use the following:

$$L^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\epsilon I_j} g_i\right)^2}{\sum_{i\epsilon I_j} h_i + \lambda} + \gamma T$$

Quality scoring function

This function returns the minimum loss value for the tree structure, so the original loss function is calculating using the most optimal weights for each leaf. Once in practice, we apply this by starting from a single root, from which we will iterate over the features and the values in each feature to evaluate every split loss reduction. The gain is calculated by calculating the loss of the father instance and subtracting the sum of the loss of both left and right branch. If the gain for the best split is negative, we stop growing the branch.

4.5 Implementation

In finance, the ability to predict accurately trends in stock prices is of great importance for investors, analysts and financial institutions in order to maximize their gains and reduce risks when investing. Machine learning algorithms have made huge leaps in recent years, specially when talking about ensemble methods such as XGBoost which as we have mentioned before is a powerful tool to tackle classification problems like the one presented in this paper about tendencies in fluctuating prices of stock belonging to Microsoft, Google, and Visa.

We aim to develop a robust predictive model by incorporating features representing momentum indicators and technical analysis metrics such as the ones mentioned in section 1 of this paper, these are:

- Relative strength index
- Stochastic Oscillator
- Williams percentage rate
- Moving Average Convergence Difference
- Price rate of change
- On Balance Value

These momentum indicators provide valuable insights about the strength and direction of the price trends, since they provide a better and more comprehensive analysis of market dynamics. Through historical stock data, momentum indicators

and the use of XGBoost we aim to provide valuable insight of the changes in stock prices within these companies.

Data Preprocessing

The initial hypothesis is that we need to take a look at the fields related to the stock prices and trading sales. In order to analyze if the prices seem to be relevant in the trading volume. Initially, we set the date field from the dataset as an index for plotting the values and to have an easier time when calculating the momentum indicators.

Once importing the data from the CSV files into our data frames, we will proceed to drop the columns that won't be necessary to train the model or to make the calculations of the momentum indicators. Also, there were no missing values in any dataset, therefore there was no need to fill in data with most frequent values or dropping any additional columns.

Original Columns	Resulting columns
● Date ● Open ● High ● Low ● Close ● Adj. Close ● Volume ● Symbol	● Date ● Close ● Adj. Close ● Volume ● Symbol

Table 4.1 Resulting columns after preprocessing

Data Sample

Date	Close	Adj Close	Volume	Symbol
2021-12-06	326.190002	320.321655	30032600	MSFT
2021-12-07	334.920013	328.894592	31021900	MSFT
2021-12-08	334.970001	328.943665	24761000	MSFT
2021-12-09	333.100006	327.107300	22214200	MSFT
2021-12-10	342.540009	336.377502	38095700	MSFT

Table 4.2 Dataset Sample

Graphs for closing prices for the three stocks

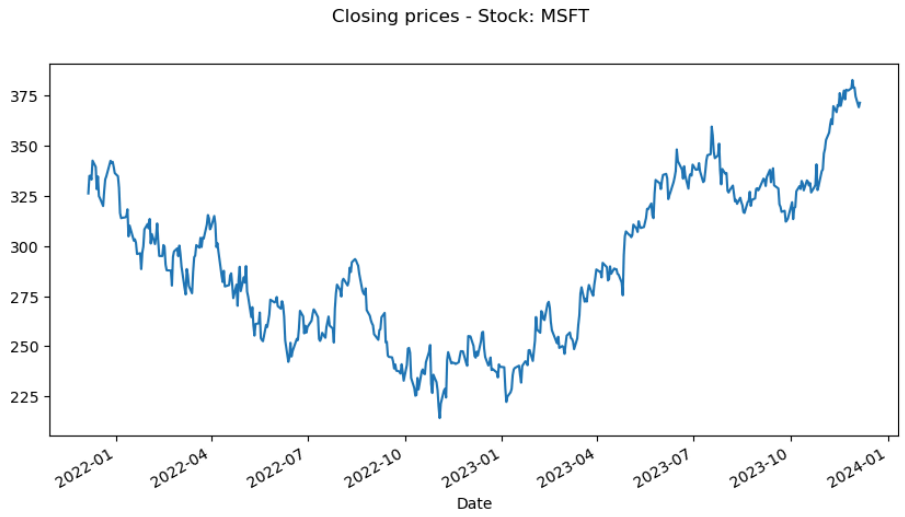


Figure 4.3 Closing prices, Microsoft 2022-2024

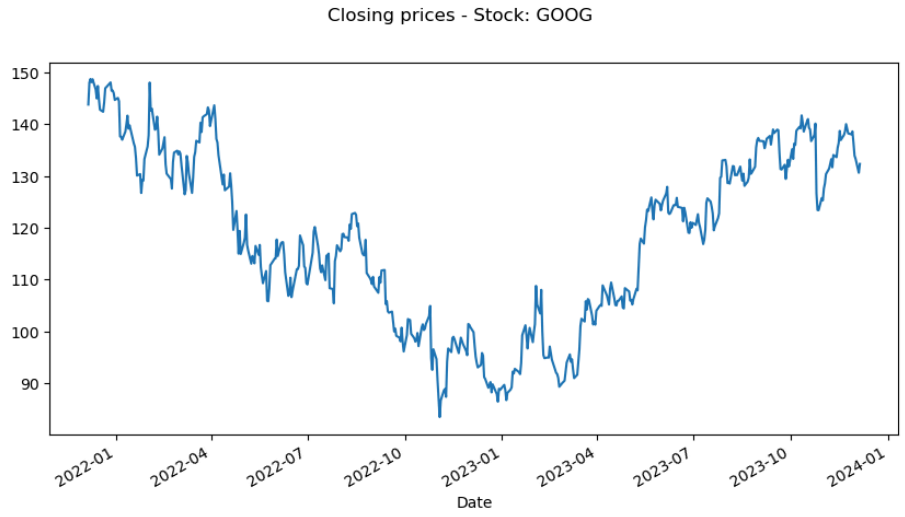


Figure 4.4 Closing prices, Google 2022-2024

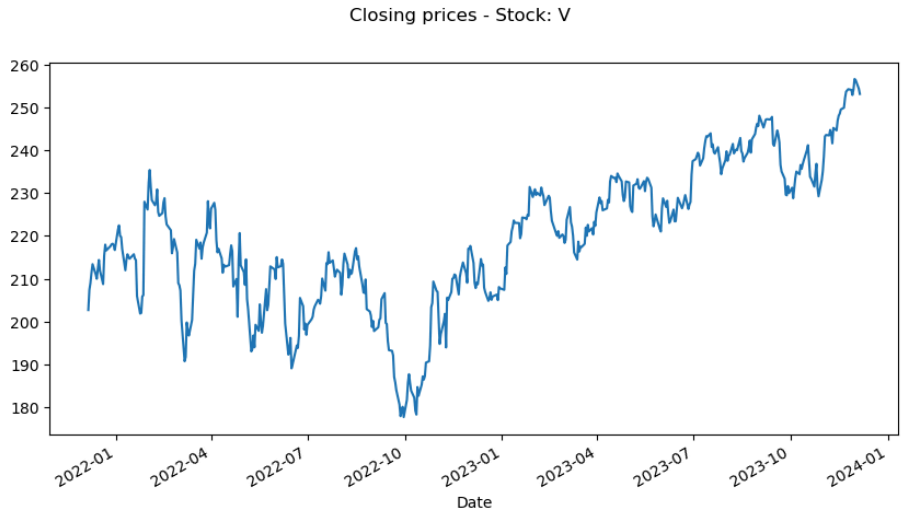


Figure 4.5 Closing prices, Visa 2022-2024

Feature Creation

We now calculate the momentum indicators that will give us buy and sell signals based on their results and the conditions specified by each one of these. The selected period for these calculations is 14 days by default, but this value can be modified if it is necessary.

Also, we create our target which represents the movement of the stock prices in definition 3 possible cases as shown before. But due to the XGBRFClassifier function only takes values starting from 0 we can't use the classes defined earlier for the random forest implementation, therefore we switched these values to 0,1 and 2 but with the same function.

Random Forest classes	XGBoost Classes
• 1 if positive • -1 if negative • 0 if no change	• 2 if positive, • 0 if negative • 1 if no change

Table 4.3 Target definition

Model Creation

Before creating the model, we set the features and target for training and testing. Also, we define a split date to create a test and a train split within our data frames all

within the function `trainModel`.

```python
def trainModel(df_train,index):
    X_train = df_train[FEATURES]
    y_train = df_train[TARGET]

    model = xgb.XGBRFClassifier()
    model.fit(
        X_train,
        y_train,
    )

    fi = pd.DataFrame(
        data = model.feature_importances_,
        index=model.feature_names_in_,
        columns=['importance']
    )

    fi.sort_values('importance').plot(kind='barh')
    plt.title(label=f'Feature importance for stock: {GROUPS[index]}')
    plt.show()

    return model
```

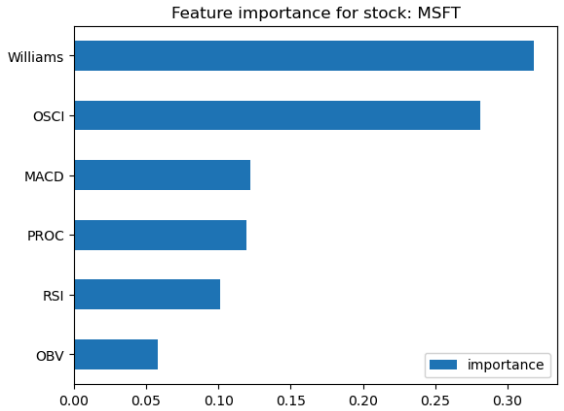


Figure 4.6 Feature Importance MSFT

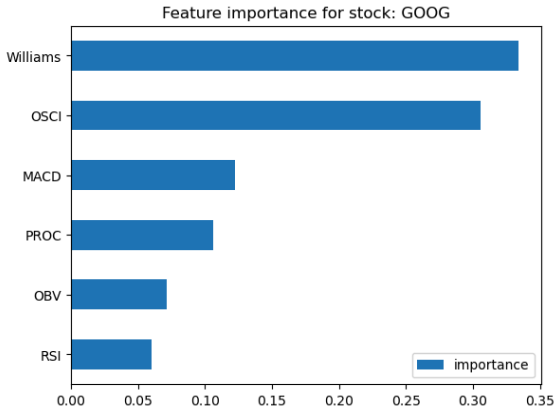


Figure 4.7 Feature Importance Google

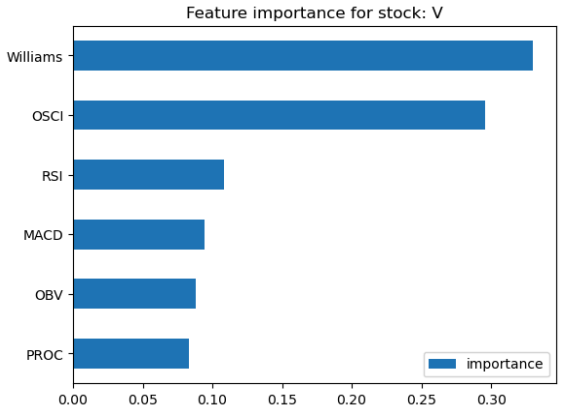


Figure 4.8 Feature Importance Visa

In figures 4.6, 4.7 and 4.8 we can see slight differences between the 3 stocks, but the 2 most important features remain the same (Williams percentage rate and stochastic oscillator). This may be due to their direct relationship with the highest and lowest closing prices within the 14-day range.

4.6 Results

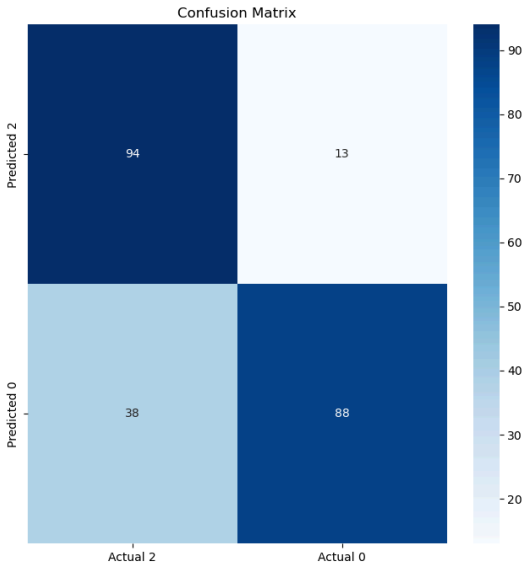


Figure 4.9 Confusion Matrix MSFT

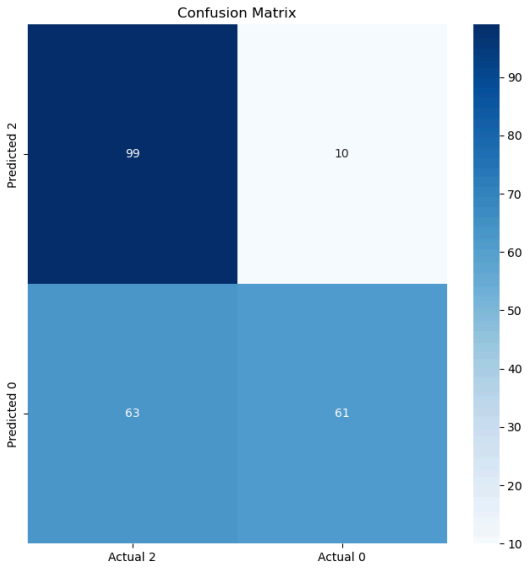


Figure 4.10 Confusion Matrix Google

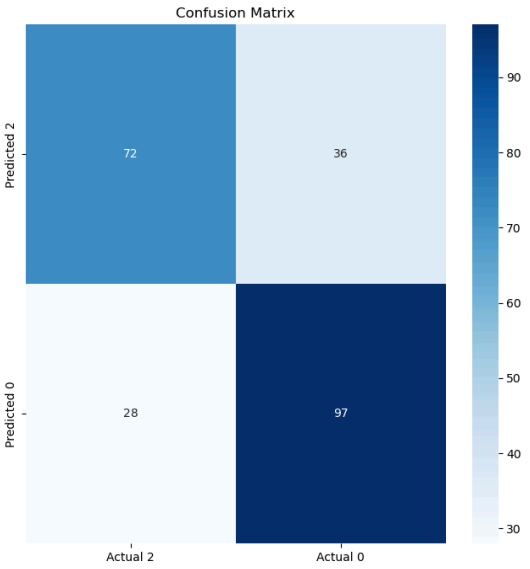


Figure 4.11 Confusion Matrix Visa

The XGBoost Random Forest Classifier obtain an accuracy score of **78.11% for Microsoft, 68.67% for Google and 72.53% for Visa**. Also letting us see how the Microsoft stock was the one that more precisely predicted true positives and negatives, similarly to the visa stock that did have a good performance but not on the same level as Microsoft stocks. On the other hand, the model applied to the Google stock had problems identifying negatives, since it had near to 60 % precision on the negative values.

4.7 Stock Market Simulation

Setting a balance of ten thousand dollars for each stock, we run the simulation script to observe how the predictions perform in a trading environment. As expected, the overall gains over time are greater within bigger time frames for each individual stock.

Days	Google	Microsoft	Visa
7	$9,899.67	$10,023.42	$10,382.73
30	$10,733.46	$9,879.87	$10,548.28
60	$11,350.49	$9,789.28	$10,356.03
90	$11,283.59	$10,026.21	$10,403.45

Table 4.4: Results of the simulation using predicted values from the XGBoost model

5. Conclusions

Presented here are the results of the stock market simulation for each algorithm. The table offers a concise and organized overview of the performance of Random Forest, XGBoost, and Long Short-Term Memory (LSTM) algorithms across Google, Microsoft, and VISA stocks.

Days	Random Forest	XGBoost	LSTM
7	$23,717.21	$30,305.81	$30,045.68
30	$49,424.82	$31,161.61	$30,920.87
60	$59,952.95	$31,495.81	$31,911.68
90	$40,115.62	$31,713.26	$33,631.61

Table 5.1. Sum of total investments.

As seen in the above table, the Random Forest algorithm makes more money in almost every iteration, except in the first 7 days when XGBoost outperforms both the Random Forest Algorithm and the LSTM model. Even though the Random Forest algorithm outperforms its counterparts, the overall accuracy shown in the model's metrics is 67.5%, in contrast, the XGBoost model holds between 68.7% to 78.1% – depending on the company – of accuracy. Although this, differences in prediction accuracy show the benefits of using XGBoost's ensemble techniques along with gradient boosting to have a more precise result in classification problems, outperforming both models in terms of accuracy and amount of misclassified positives and negatives.

Despite XGBoost demonstrating higher accuracy rates ranging from 68.7% to 78.1% across various stocks compared to Random Forest's 67.5%, it's noteworthy that Random Forest emerged as the more profitable model in this particular stock market simulation. This outcome could be attributed to several factors. Firstly, Random

Forest's robustness to noise and outliers might have allowed it to better generalize the underlying patterns in the stock market data, resulting in more stable and profitable predictions.

Additionally, Random Forest's inherent capability to capture non-linear relationships between predictor variables and stock price movements could have provided it with an edge in accurately predicting market trends, even with slightly lower accuracy metrics. Moreover, the ensemble averaging approach of Random Forest, which aggregates predictions from multiple decision trees, likely helped mitigate the risk of overfitting and improve generalization performance, ultimately leading to higher profitability.

Lastly, the specific characteristics of the dataset, such as the distribution of features and the nature of the relationship between predictors and the target variable, might have favored Random Forest's strengths over XGBoost, resulting in its superior performance in terms of profitability despite slightly lower accuracy rates. Overall, while XGBoost excelled in accuracy metrics, Random Forest's robustness, interpretability, and ability to capture complex relationships proved advantageous in maximizing profitability in this stock market simulation scenario.

Placing second at almost every iteration of the simulation, the LSTM model, as seen throughout the **figures 2.8 to 2.10**, follows the actual stock prices closely and within the simulation, it performs as expected considering how the sudden initial spike affects the model and the curve (as discussed in **section 2.3. LSTM in Action**, **subsection 2.3.4. Results**).

As future improvements, a potential enhancement involves increasing the dataset with a more extensive time period, to further enrich the models' training process. By expanding the amount of data, the models may gain a more comprehensive understanding of market trends and fluctuations, furthermore refining the models' predictive capabilities.

6. Bibliography

1. *Basak, S., Kar, S., Saha, S., Khaidem, L., & Roy Dey, S. (2019). Predicting the direction of stock market prices using tree-based classifiers. The North American Journal of Economics and Finance, 47, 552-567. https://doi.org/10.1016/j.najef.2018.06.013*
2. *What are Neural Networks?. IBM*. (n.d.). https://www.ibm.com/topics/neural-networks
3. *What are Recurrent Neural Networks?.* IBM. (n.d.). https://www.ibm.com/topics/recurrent-neural-networks
4. GfG. (2023, December 8). Deep Learning introduction to long short term memory. GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/
5. Siami-Namini, S., & Namin, A. S. (2018). Forecasting Economics and Financial Time Series: ARIMA vs. LSTM. arXiv (Cornell University).https://doi.org/10.48550/arxiv.1803.06386
6. Singh, A. (2023, 22 noviembre). *A Comprehensive Guide to Ensemble Learning (with Python codes)*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/#:~:text=Ensemble%20learning%20is%20a%20machine,collective%20intelligence%20of%20the%20ensemble.
7. *Supervised Machine learning - Javatpoint*. (s. f.). www.javatpoint.com. https://www.javatpoint.com/supervised-machine-learning
8. *What is XGBoost?* (s. f.). NVIDIA Data Science Glossary. https://www.nvidia.com/en-us/glossary/xgboost/
9. *XGBoost Documentation — xgboost 2.0.3 documentation*. (s. f.). https://xgboost.readthedocs.io/en/stable/
10. GfG. (2023, 31 marzo). *Gradient Boosting in ML*. GeeksforGeeks. https://www.geeksforgeeks.org/ml-gradient-boosting/
11. Chen, T., & Guestrin, C. (2016). XGBOOST: a scalable tree boosting System. En *ACM Digital Library*. https://doi.org/10.1145/2939672.2939785